THE UNIVERSITY OF
MELBOURNE

Department of
Computer Science and Software Engineering

Test Plan

for

Managing the analysis of massive DNA sequence data

Version: 3.0
26th August 2009

The purpose of this Test Plan is to show how the system may be tested. This document provides detailed steps and strategies that are employed to ensure each requirement can be fulfilled. The Test Plan also includes a plan for system level tests, and strategies for testing the system at a module level. As the project develops, the document will be revised and modified.

Copyright notice

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria
AUSTRALIA
3010

ICT Building
111 Barry Street
Carlton

Tel: +613 8344 1300
Fax: +613 9348 1184
https://www.csse.unimelb.edu.au/

Version: 3.0
26th August 2009.
https://www.cs.mu.oz.au/SE-projects/s340gd

Credits

This document was written by members of Team D

CONTENTS

ONE

Introduction

1.1 Scope of the Document

This document will explore test cases for each of the requirements in order to prove that they can be satisfied. It will also detail how the team will approach their methods of testing.

The document explores acceptance test items, which are separated based on the two program flows - Pre-processing Flow and BLAST flow. Each of the two sections is divided into functional and non-functional requirements. Each subsection is then categorised into essential and optional requirements. Under each requirement, there is a description of a test case for it. For some requirements, technical details are also included after the descriptions for clarification.

In addition to that, a test plan is also written to test each main module of the program. The aim of this Module Level Test Plan section is to ensure that each module is testable to ensure that it is correct and complete.

The document also includes future test plans, which lists the tests that will be written to test at system level. Furthermore, strategies of how the development team will test the software is also described in the document.

TWO

Acceptance Test Items

2.1 Functional Requirements

This section documents acceptance tests for functional requirements that has been stated in the Software Requirements Specification (SRS). The test describes steps that will be executed to ensure the requirement is satisfied.

Pre-processing Flow

2.1.1 Essential

This section contains acceptance tests for essential functional requirements for the pre-processing flow that must be satisfied by the end system in order for it to be considered correct and complete. These tests are essential and must be executed by all means.

Input Files Stage

2.1.1.1 Input files selection (Requirement 4.1.1.1)

At the screen to select input files, there should be a button to press to launch a file chooser to select an input file. If a paired end file exists for the input file, the user may specify this and then choose to select it in the same way using another similar button.

Test Case:

1. At the screen to select input files, there should be a button to press to launch a file chooser to select an input file. If a paired end file is required, another button may be pressed to select another input file after a checkbox has been ticked.
2. The user will see a screen with a button. When the button is pressed, a file chooser will appear, where the user can select an input file in TXT format.
3. After the user has selected the file, he gets to choose a paired end file, if it exists. This is done the same way as before, but with another button below the previous one. If only one input file is needed, skip this step.

4. The system should now be able to output the contents of all the selected input files into separate files.
5. The contents of the newly generated files will be exactly the same as the contents of the selected files.
6. This comparison can be done using UNIX command 'diff'.

2.1.1.2 Default selection of short read from input file(s) (Requirement 4.1.1.2)

Once input file(s) are selected, a display of the first twenty short-reads it contains will be shown. The system will automatically select the 6th colon separated string in each line of the input file(s) to be the short read.

Test Case:

1. The short read of the first line in the file is displayed on the screen. This will be the same as the 6th column of the first line of the file.
2. This verification will be done by eye.

2.1.1.3 Default selection of quality from input file(s) (Requirement 4.1.1.3)

The seventh column of any line of input will represent the quality of that particular line's short read.

Test Case:

1. The quality of the first line of the input file will be printed out.
2. This output will then be compared to the seventh column of the first line of the input file by eye.
3. The two strings must be the same.

Quality Processing Stage

2.1.1.4 Specification of quality threshold for each input file (Requirement 4.1.1.4)

There will be a screen where the user can input values for the quality threshold. The system will only contain short reads that are above the specified quality threshold.

Test Case:

1. The user will be able to enter in values for the quality threshold.
2. The user may alter values for the minimum ASCII sum and the minimum ASCII character.
3. Sequences that are below the either thresholds will be removed from the output.

2.1.1.5 Saving input files that have been filtered with sufficient quality (Requirement 4.1.1.5)

Once quality processing has been done, a new version of each input file will be ready to be saved. This new version will have lines with short reads of insufficient quality removed. Three test cases have been written for this particular requirement. The system must pass all three of the following test cases for this requirement to be met.

Test Case #1:

1. The user will specify the quality threshold to be lower than the minimum quality value from all the lines of input.
2. After quality processing is done, the new version of input file is saved.
3. This new input file will still contain the same number of lines as before, with no alterations whatsoever.

Test Case #2:

1. The user will specify the quality threshold to be higher than the maximum quality value from all the lines of input.
2. After quality processing is done, the new version of input file is saved.
3. This new input file will contain no lines of input at all.

Test Case #3:

1. The user will specify the quality threshold to be any value within between the minimum and the maximum quality value from all the lines of input.
2. After quality processing is done, the new version of input file is saved.
3. This new input file will still only contain lines with quality at least the quality threshold specified.

Grouping Stage

2.1.1.6 Region selection for each short read's identification (Requirement 4.1.1.6)

Once input file(s) are selected, a display of the first twenty short reads it contains will be shown. The user will be able to indicate a column of characters of the short reads that represent the unique ID for the sequences. Many short reads may have identical identification because they are meant to be in the same group. Let's say the selected characters for rows 1 to 5 are: 'ACGT', 'AGGT', 'GACT', 'AGGT' and 'TAGC' respectively, each of these set of characters will be the identification for each of the rows. Therefore, row 2 and row 4 will have the same ID, meaning that they are related in some way.

Test Case #1 (no paired end file):

1. The short reads should be displayed on screen.
2. The user is given the choice to indicate a column of characters of the short reads that represents the identification of that short read.
3. The user will select columns 10 to 13 (inclusive) using the mouse. (Columns start from 0.)
4. After clicking a confirmation button, the characters from columns 10 to 13 (inclusive) of each row will be the identification for that row's short read.

Test Case #2 (paired end file exists):

1. The short reads of both files should be displayed on screen side by side.
2. The user is given the choice to indicate a column of characters of the short reads that represents the identification of that short read.
3. The user will select columns 10 to 13 (inclusive) of the first input file using the mouse. (Columns start from 0.)
4. After clicking a confirmation button, the characters from columns 10 to 13 (inclusive) of each row of the first input file will be the identification for that row and also its corresponding row's short read.

2.1.1.7 Wildcards (Requirement 4.1.1.7)

Users will be able to specify a sequence of input with wildcards to be grouped.

Test Case:

1. A sequence of input with wildcards, AG*T is to be grouped.
2. In the output screen, any input sequences of AGCT, AGAT, AGGT or AGTT, if any should be grouped.

2.1.1.8 Grouping of input sequences (Requirement 4.1.1.8)

After the user has selected identifications for each string, the user will be able to place the identifications into groups. There will be a minimum of two groups.

Test Case:

1. Two textfields will be displayed on screen where each textfield symbolizes a group.
2. Additional textfields may be added at a click of a button.
3. The user is given the choice to put each identification into groups.
4. When grouping, the user will enter ”AACT,AGGT,GACT” in the textfield.
5. The software should alert the user if an identification is in two groups.
6. After clicking the next button, the identifications will be stored in groups on the system.
7. Files will be created on the local machine with a list of short reads belonging to each group in each file.
8. The short reads in each file will each contain an identification that is user-specified to belong to the group the file represents. itemThis can be validated by eye.

2.1.1.9 Saving grouped files that have been generated from grouping (Requirement 4.1.1.9)

Files that represent each group will be generated, containing all the identifications specified for that particular group.

Test Case:

1. Group 1 is given the identifications ’ACCT’ and ’AGCT’.
2. Group 2 is given the identification ’A*TG’.
3. Group 3 is given the identifications ’CACT’, ’TAGT’ and ’TCCA’.
4. A file will be generated containing the strings ’ACCT’ and ’AGCT’.
5. A second file will be generated containing the strings ’AATG’, ’ACTG’, ATTG’ and ’AGTG’.
6. A third file will be generated containing the strings ’CACT’, ’TAGT’ and ’TCCA’.
7. Each string is on a new line.

2.1.1.10 Tag file generation (Requirement 4.1.1.10)

A tag file that contains the number of times each short read present in the input file(s) will be generated after requirement 3.1.6.

Test Case:

1. For example, group 1 can contain the identifications 'ACGT', 'AATG' and 'CTGA'.
2. An example input file has 20 short reads with 'ACGT', 50 short reads with 'AATG' and 5 short reads with 'CTGA' as identification.
3. After requirement 4.1.3.7, a tag file is generated.
4. A tag file with the filename 'ACGT' will be generated. It will contain the BLAST regions of all unique short reads with the identification 'ACGT'. The number of times each short read occurs in the original input file(s) will also be specified.
5. Similarly, a tag file with the filename 'AATG' will be generated, containing the BLAST regions of all unique short reads with the identification 'AATG'. The number of times each short read occurs in the original input file(s) will also be specified.
6. Finally, a tag file with the filename 'CTGA' will also be generated, containing the BLAST regions of all unique short reads with the identification 'CTGA'. The number of times each short read occurs in the original input file(s) will also be specified.

2.1.2 Optional

This section documents acceptance tests for functional requirements for the pre-processing flow that are requested by Jason but considered optional. These tests are useful to the development team but are of low priority compared to tests for essential functional requirements. These tests will only be executed if the optional functional requirements are implemented in the program.

Input Files Stage

2.1.2.1 Check for conflicting identifications while grouping (Requirement 4.1.2.1)

If an identification exists in more than one group, an alert message will be shown.

Test Case #1:

1. Group 1 will be given the identifications 'ATTG' and group 2 will be given the identifications 'GATG'.
2. No alert message will be given.

Test Case #2:

1. Group 1 will be given the identifications 'ATTG' and group 2 will be given the identifications 'GATG' and 'ATTG'.
2. An alert message will be given.

Test Case #3:

1. Group 1 will be given the identifications 'GAT*', group 2 will be given the identifications 'GATG'.
2. An alert message will be given.

BLAST Processing Flow

2.1.3 Essential

This section contains acceptance tests for essential functional requirements for the BLAST processing flow that must be satisfied by the end system in order for it to be considered correct and complete. These tests are essential and must be executed by all means.

Input Files Stage

2.1.3.1 Input files selection (Requirement 4.1.3.1)

At the screen to select input files, there should be a button which launches a file chooser when pressed. The user may select only one input file in TXT format.

Test Case:

1. At the screen to select input files, a file chooser is launched when a button is pressed.
2. The user will select an input file of TXT format.
3. The input file is stored in the system to be accessed later on.

2.1.3.2 Default selection of short read from input file(s) (Requirement 4.1.3.2)

Once input file(s) are selected, a display of the first twenty short reads it contains will be shown. The user will be able to indicate a column of characters of the short reads that represent the unique ID for the sequences.

Test Case:

1. The short reads should be displayed on screen.
2. The user is given the choice to indicate a column of characters of the short reads that is to be used as unique ID.
3. The user will select columns 10 to 13 (inclusive) using the mouse.
4. After clicking a confirmation button, the characters from columns 10 to 13 (inclusive) of each row will be unique ID for that row of the short read.
5. Let's say the selected characters for rows 1 to 5 are: 'ACGT', 'AGGT', 'GACT', 'AGGT' and 'TAGC' respectively, each of these set of characters will be identification for each of the rows.
6. Therefore, row 2 and row 4 will have the same ID, meaning that they are related in some way.

2.1.3.3 Region selection for BLASTing input sequences (Requirement 4.1.3.3)

After the user has chosen a unique identification for each sequence, the user will be presented with a screen where they must select the region to be BLASTed. The mouse will be used to select the region.

Test Case:

1. The short reads should be displayed on screen.
2. The user is given the choice to indicate a column of characters of the short reads that are to be BLASTed.
3. The user will select columns 10 to 18 (inclusive) using the mouse.
4. When the first row is highlighted, the corresponding columns in the rows below are also highlighted.
5. After clicking a confirmation button, the characters from columns 10 to 18 (inclusive) of each row will be stored while the rest is treated as junk data and can be discarded.

Reference Files Stage

2.1.3.4 Reference files selection (Requirement 4.1.3.4)

At the screen to select reference files, there should be a button to press to launch a file chooser to select a reference file. If more reference files are required, a button may be pressed to select another reference file. The system will allow up to ten reference files to be selected.

Test Case:

1. At the screen to select reference files, there should be a button that launches a file chooser for selecting a reference file.
2. The user will see a screen with a button. When the button is pressed, a file chooser will appear, where the user can select a reference file in CSV format.
3. After the user has selected the file, he gets to add up to ten reference files as necessary. These files can be selected as described in steps 1 and 2.
4. The user should now have selected one or up to ten reference file(s) as necessary.

2.1.3.5 Output column selection (Requirement 4.1.3.5)

Once the reference files have been selected, the GUI will allow the user to select the columns from the reference files that the output table will have. This will be done by clicking on the checkboxes beside a vertical list of the columns that the user wants to see in the output.

Test Case:

1. For example, the reference file has the columns 'Oligo.ID', 'Catalog Number', 'Library Name', 'Plate', 'Full Hairpin Sequence', and 'Sense Sequence'.
2. After this file is opened, the user will see a screen with all the columns in a vertical list.
3. The list will contain the following items: 'Oligo.ID', 'Catalog Number', 'Library Name', 'Plate', 'Full Hairpin Sequence', and 'Sense Sequence'.
4. A checkbox will be present beside each item.
5. The user ticks the checkboxes that correspond to the items they want to be used (e.g. 'Oligo.ID' and 'Full Hairpin Sequence') as column headings in the output.
6. Once this is done, the output of the current run of the program will contain the column headings 'Oligo.ID' and 'Full Hairpin Sequence'.

2.1.3.6 Region selection for BLASTing reference files (Requirement 4.1.3.6)

Once a reference file(s) is/are open, a display of the first twenty lines of short reads it (the first file) contains will be shown. The user will be able to indicate a column of characters of the short reads that will be used for BLASTing.

Test Case:

1. The first twenty lines of short reads should be displayed on screen.
2. The user is given the choice to indicate a column of characters of the short reads that will be used for BLASTing.

3. The user will select columns 10 to 13 (inclusive) using the mouse.
4. After clicking a confirmation button, the characters other than those from columns 10 to 13 (inclusive) of each row will be treated as junk data and ignored during the BLAST process.

BLAST Stage

2.1.3.7 Setting and changing parameters of the BLAST process (Requirement 4.1.3.7)

Before beginning the BLAST process, the user may change parameters that affect how BLAST is run. The user will be able to input numeric values for parameters such as the expected failure margin(-e), window size(-W), the number of hits(-k), the query strands to search against database(-S), the number of database sequences to show one-line descriptions(-v) and the number of database sequences to show alignments(-b).

Test Case:

1. The user is presented with a screen where they may input numeric values for the expected failure margin, the window size, the number of hits, the query strands to search against database, the number of database sequences to show one-line descriptions and the number of database sequences to show alignments.
2. After the user has pressed 'Next', the system will store the variables and use them later in the system call to BLAST.

2.1.3.8 Tag file generation (Requirement 4.1.3.8)

A tag file that contains the number of times each short read present in the input file(s) will be generated after requirement 3.1.6.

Test Case:

1. For example, group 1 can contain the identifications 'ACGT', 'AATG' and 'CTGA'.
2. An example input file has 20 short reads with 'ACGT', 50 short reads with 'AATG' and 5 short reads with 'CTGA' as identification.
3. After requirement 4.1.3.7, a tag file is generated.
4. A tag file with the filename 'ACGT' will be generated. It will contain the BLAST regions of all unique short reads with the identification 'ACGT'. The number of times each short read occurs in the original input file(s) will also be specified.
5. Similarly, a tag file with the filename 'AATG' will be generated, containing the BLAST regions of all unique short reads with the identification 'AATG'. The number of times each short read occurs in the original input file(s) will also be specified.
6. Finally, a tag file with the filename 'CTGA' will also be generated, containing the BLAST regions of all unique short reads with the identification 'CTGA'. The number of times each short read occurs in the original input file(s) will also be specified.

2.1.3.9 FASTA formatted files (Requirement 4.1.3.9)

The input file(s) and the reference file(s) must be converted to FASTA format before being run through BLAST.

Test Case:

1. After conversion to FASTA, BLAST should accept the converted input file(s) and the converted reference file(s) as input to be BLASTed.
2. If BLAST completes successfully, that means the files have been converted to the right format, as BLAST only accepts FASTA files as inputs.

2.1.3.10 Preview of results (Requirement 4.1.3.10)

The first set of results will be displayed as soon as possible. This is to allow the user to see if they have made any mistakes.

Test Case:

1. Once the user has started to BLAST, output should be displayed only for the first set of files to be BLASTed.
2. Output should be displayed within one minute of pressing the BLAST button.

2.1.3.11 Halt BLAST (Requirement 4.1.3.11)

Once processing, BLAST should be able to be stopped.

Test Case:

1. The task manager should show BLAST running when it's processing.
2. When the button used for halting the BLAST process is pressed, a confirmation box should appear.
3. On confirmation, a dialogue box should pop-up, explaining that the BLAST process has stopped prematurely.
4. The BLAST process should now be gone from the task manager.

2.1.3.12 Multiple threads (Requirement 4.1.3.12)

The inputs into BLAST should be processed in multiple threads concurrently.

Test Case #1:

1. As each thread is finished processing, the resulting output should be available to be displayed.
2. Therefore, the user will be able to see the results as each thread has been processed.

Test Case #2:

1. BLAST is first executed with all data in one thread.
2. The time needed to finish processing all the data, T1, should be noted down.
3. BLAST is then executed once more with all the data broken up into several threads (five, for instance).
4. The time needed to finish processing all the multiple threads of data, T2, should be less than the time needed to process all the data together in one thread, T1.
5. If T2 < T1, then the test case is successful.

Output Stage

2.1.3.13 Tabular Output Display(Requirement 4.1.3.13)

The BLAST output must be displayed in tabular format.

Test Case:

1. The output from BLAST produces a text file with output in tabular format.
2. Output in the file will be printed out on the console.
3. The results will be seen in an output table.

Features

2.1.3.14 Read in files properly for display (Requirement 4.1.3.14)

Files should be read in correctly for display. If they are not, an error message must appear.

Test Case:

1. While choosing a file (whether it be an input or reference file), files that are not of CSV format will not be visible.
2. If a file is not successfully read, an error message will alert the user.
3. If a file is read in properly, its first 20 lines of contents will be visible. If a file has less than 20 lines, all the available lines will be shown.

2.1.4 Optional

This section documents acceptance tests for functional requirements for the BLAST processing flow that are requested by Jason but considered optional. These tests are useful to the development team but are of low priority compared to tests for essential functional requirements. These tests will only be executed if the optional functional requirements are implemented in the program.

2.1.4.1 Pause BLAST (Requirement 4.1.4.1)

Once processing, BLAST should be able to be paused and then resumed.

Test Case:

1. The task manager should show BLAST running when it's processing.
2. When the button used for pausing the BLAST process is pressed, the CPU Usage in the task manager will decrease by more than 5%.
3. When the button used for resuming the BLAST process is pressed, the CPU Usage in the task manager will rise back to roughly the previous value.

2.1.4.2 Real-time update of results (Requirement 4.1.4.2)

Output from BLAST shall be displayed on screen immediately when available.

Test Case:

1. Once the user has pressed the BLAST button, output from BLAST will be displayed immediately.
2. The screen will update every 10 seconds with new additions appended to the list.

2.1.4.3 Saving output (Requirement 4.1.4.3)

The BLAST output is in TXT format, and this should be able to be saved to a CSV file.

Test Case:

1. The output from BLAST is known to be in TXT.
2. At the end of the BLAST process, the user will be able to save the output as a CSV file.
3. If this file can be opened and read by Microsoft Excel, then the test case is successful.

2.2 Non-functional Requirements

This section documents acceptance tests for non-functional requirements that has been stated in the Software Requirements Specification (SRS). The test describes steps that will be executed to ensure the requirement is satisfied.

2.2.1 Essential

This section contains acceptance tests for essential non-functional requirements that must be satisfied by the end system in order for it to be considered correct and complete. These tests are essential and must be executed by all means.

2.2.1.1 Resource Constraint - System is not CPU intensive (Requirement 4.2.1.1)

The BLAST process should take enough computer processing power while not hindering the user doing other tasks.

Test Case:

1. Firefox and Microsoft Excel are run less than five seconds apart from each other. The time taken for both of them to open up is recorded down as T1.
2. Both programs are then closed.
3. While the BLAST process is executing, Firefox and Microsoft Excel are run. The time taken for both of them to open up is recorded down as T2.
4. T2 should be no more than 20 seconds greater than T1.

2.2.1.2 Usability - Simple to use (Requirement 4.2.1.2)

Testers can be picked out from a group of people. The results from surveying 5 different testers should indicate whether the program is simple to use or not.

Test Case:

1. Five random University of Melbourne undergraduate students will be chosen as testers.
2. Each of them will use the program under the monitoring of the development team and the client.
3. Each of them will be told the unique ID used for grouping input sequences (which the main users are assumed to know) during the stage of the program for grouping input sequences.
4. If every tester takes less than 10 minutes to begin the BLAST process, the program is simple to use.

2.2.1.3 Platform Compatibility - Cross platform support (Requirement 4.2.1.3)

The program will be executable on Windows and Macintosh machines.

Test Case:

1. The program is installed on a Windows and a Macintosh machine.
2. After the installation is done, the program is run.
3. Each feature of the program is tried out at least twice.
4. The GUI interface between the two machines are compared side by side.

2.2.2 Optional

This section documents acceptance tests for non-functional requirements requested by Jason but considered optional. These tests are useful to the development team but are of low priority compared to tests for essential non-functional requirements. These tests will only be executed if the optional non-functional requirements are implemented in the program.

2.2.2.1 Extensibility - Future algorithm support (Requirement 4.2.2.1)

The software will use the BLAST process without altering it.

Test Case:

1. The system should not have to alter any parts of BLAST at all.

THREE

Strategies

3.1 Approach

This section explains about the sort of strategy to approach testing the system.

3.1.1 Event Flow Graphs

Event flow graphs represent events and events interaction of the software system. Creating event flow graphs helps the team to represent all possible event sequences that can be executed on the GUI. By keeping track of each event call to the system, it is possible to identify the problem occur during the execution of the GUI software.

3.1.2 Code Review/Inspections

A testing team separate from the design and coding team will carry out the code review. With a chart of requirements, the team tests each function whether the function meets the requirements or not. The columns of the chart should have all the functional requirements and its identities and the rows should have methods of testing. The team will be aided by tools such as Hudson, Checkstyle and JUnit tests.

3.1.3 Coverage-Based Testing

The testing team will mostly write test cases to test statement coverage and path coverage since these two criterias are important as every statement and execution path of the program should be exercised at least once. By creating a control-flow graph, the testing team can easily choose test cases to satisfy the coverage criteria.

3.1.4 Boundary Analysis

We will use boundary analysis on region selection for BLASTing to make sure the selected region is what the user wanted.

3.1.5 Black Box Testing

3.1.5.1 Alpha Testing

A group of 10 non-computer science students from the university will be invited to use the system. They are expected to express their thoughts as they carry out each particular input or action. Any type of abnormal behaviour of the system is noted and will be rectified by the developing team.

3.1.5.2 Functional Testing

The testing team will test the system for the required functional requirements. Tests are written based on the SRS to check whether the system behaves as expected.

3.1.5.3 Usability Testing

Previously mentioned in Alpha Testing, the group of selected students will test the system for its user friendliness. After the testing has completed, they will be asked to rate the system based on a scale of 1 to 10, 1 being "Terrible" and 10 being "Great" and state the reason why do they think so. Depending on the feedback, the developing team may have to readjust the system.

3.2 Integration Strategy

This section tells about what integration strategy will be implemented and the reason why it was selected to be used to integrate the subsystems into the system.

A bottom-up integration strategy is chosen to implement and integrate the modules. The lowest-level modules are implemented first before the modules at the next level up are implemented to form the subsystems and so on until the whole system is complete and integrated.

These modules are considered as the lowest-level modules:
Reference File
Input File
Quality File

The mid-level modules are:
CSV Reader
Update and Request Data
FASTA File
Tag File
CSV File
Short Read Comparator

These are the high-level modules that the team has identified:
PreProcessing Transformer
Parameters
CSV Writer
BLAST Evoker
Results

There are a few advantages for using this method as an integration strategy. Firstly, this method is common for object-oriented programs because testing and integration starts at base classes and then integrating the classes that depend on the base classes and so on. Besides that, the behaviour of each interaction points are crystal clear as components are added in a controlled manner and test repetitively.

This strategy requires test drivers to be written to supply input data to another piece of code. The modules being tested need to have input data supplied to them via their interfaces. The driver program typically calls the modules under test supplying the input data for the tests. However, it may be difficult to write and maintain test drivers if the coding team is unable to tell whether the code they have written comes off as complex or confusing.

3.3 Strategies For Testing Modules

This section describes the strategies that will be used for testing certain modules.

3.3.1 File Reader

A comparison technique will be used to check whether a File is read in properly by the File Reader or not. This can be done using the Unix command 'diff' on the contents of File read into the system with the actual contents of the file on the local machine.

3.3.2 Parameters

The Netbeans debugging tool will be used to check that the parameters are read in properly from the view.

3.3.3 CSV Writer

This module will be treated as a black-box. This is an open source code so we can assume that it is accurate and correct. However, it is best to ensure that the code is right. Therefore, the BLAST output will be fed into the black-box, and the output from the black-box will be compared with the standard BLAST output. This will be done by inspection, and Microsoft Excel 2003 can be used to open the file just to make sure that it is saved properly as a CSV file.

3.3.4 Results

To test whether the Results module reads and displays the CSV file correctly, the testing tool FEST will be used to test the user interface.

3.3.5 PreProcessing Handler

A black-box testing approach will be used for testing this module. The tag files, FASTA files, grouped files and quality files generated have to be of the correct content. The contents of these files will be verified by inspection.

3.3.6 BLAST Processing Handler

A white-box testing approach will be employed. The Netbeans debugging tool will be used to check if attributes that are to be passed into BLAST are correct. Code inspection will also be carried out. Errors should be easily spotted because if the files passed into BLAST are in the wrong format or if the parameters are invalid, BLAST would not run correctly.

FOUR

Tools and Test Schedule

4.1 Tools and Support Structures

This section describes the tools and support structures that the team employ in order to thoroughly and regressively test the system. These tools are discovered through the intensive research on books specifically the Java Power Tools book written by John Ferguson Smart and also on the internet.

4.1.1 JUnit

JUnit 4 is used for writing unit test cases to ensure the system classes function correctly. It is widely used by software companies and has many extensions for more specialised testing. Being simple, easy and flexible, the unit test cases written using JUnit can be executed in isolation and not depend on any tests cases to be run beforehand. JUnit tests can be used in Ant to run automated testing.

4.1.2 FEST

FEST is a testing framework for testing GUIs. It is easy to create and maintain large functional GUI tests since a GUI ”unit” can be made up of more than one component with each of them enclosing more than one class. To create thorough functional GUI test, FEST helps to simulate user events, provide a reliable mechanism for finding GUI componenets and tolerate changes in a component’s position and/or layout. FEST also has features to support JUnit 4 tests and create screen shots of failing tests which can be embedded in a HTML test report when using JUnit. Using FEST to test GUIs can make the entire system safer and more robust.

4.1.3 JConsole

JConsole is a JMX-compliant graphical tool for monitoring a Java virtual machine. It aids in monitoring applications and identifying performance issues such as keeping an eye on memory-consumption over a long period and detecting memory leaks. This tool is essential to the team to monitor thread activity and detecting deadlocks.

4.1.4 Hudson

Hudson is an extensible continuous integration engine which else monitor, test and build our software project. It will integrate changes to the project, integrate with Subversion and JUnit, and alert the team via email on the failure of a build.

4.1.5 Ant

Ant is a tool which will assist in the building of the project. It is very similar to make and Makefiles, but with XML for portability. This will be much easier to use rather than using the builder from within NetBeans.

4.1.6 Cobertura

Cobertura is a coverage testing tool, especially for Java. Cobertura allows us to identify which parts of the software are being executed the most, and which parts aren't being run at all. This will assist us in writing better test cases and optimizing the code in the correct places.

4.1.7 Trac

Trac will be used to issue bug reports and suggestions for fixing the system based on the tests that have been carried out. These will be done using the ticketing system. Each ticket has a comment section that will be used for discussion on the particular bug or fix.

4.1.8 Checkstyle

Checkstyle is an open source tool that can help enforce coding standards and best practices, with a particular focus on coding conventions. It can be used as a standalone tool to generate reports on overall code quality, and can also be incorporated into the developer's work environment, providing real-time visual feedback about the code being written. It integrates well with Hudson and the IDE NetBeans which helps the team.

4.1.9 FindBugs

FindBugs is a static analysis tool to look for bugs in Java code. This is used to identify problem patterns in code such as bad practices, correctness, and security vulnerability. The tool is actively developed and can be used as a plugin in NetBeans and Hudson.

4.2 Test Schedule

Unit tests will be written based on the Prototype. This task begins on the 29th of August. This task will carry on for five days and then integration tests will be written. As integration tests are being written, unit tests will be carried out on the Prototype (which will begin its transformation into the final product). This will carry on for five days and integration tests will then be carried out. Usability testing will be done a week into the mid-semester break. This is also right after the implementation stage is meant to finish.

A

Glossary

The following definitions, acronyms and abbreviations are used throughout this document:

1. **CLI:** Command Line Interface
2. **CSSE:** Department of Computer Science and Software Engineering
3. **GUI:** Graphical User Interface